SFA Modernization Partner

United States Department of Education
Student Financial Assistance

Internet Application and Technical Architecture Standards

Task Order #4
Deliverable #4.1.5

February 16, 2000

Table of Contents

1 Scope of this Document

"Analyze the proposed standards and document them in such a way as to promote a clear understanding by the intended audience and clear judgment on compliance. Standards will be documented according to prevailing GAO/OMB criteria. Set objectives for compliance by understanding constraints that may inhibit the adoption of the standards by existing and in-process systems." -- *Task Order 4 Technical Proposal Task Approach*

"Application and technical architecture standards documented in Microsoft Word ready for release to SFA and contractors including targets/objectives for compliance from non-compliant systems. These standards will be tested for readability and understandability by SFA and contractor personnel." -- *Task Order 4 Technical Proposal Acceptance Criteria*

The primary purpose of this document is to describe the SFA's standard application environment at a level that is meaningful to enterprise and application architects. The application environment is defined by the standards SFA has selected to utilize for all new software development from this point forward. Specifically, SFA has decided to embrace an Internet-centric model of application development that is based on the Java 2 Enterprise Edition language and APIs (application programming interfaces). This document assumes the reader is familiar with the Java platform's concepts of code portability via the Java Virtual Machine and the Core APIs. For more information on these issues, see the on-line introduction at <http://java.sun.com/nav/whatis/>.

In describing the environment, this document makes reference to the larger framework of standards, policies, and procedures in which the application development process exists. However, this framework is extensively described in other documents, even though some of those documents have not been released yet. The goal in mentioning this larger framework is to help architects to understand the entire process and where these application standards fit, not to fully describe these other parts.

Lastly, readers should not consider this document to be a static manifesto. The Enterprise Architecture Team is keenly interested in hearing from practitioners about what works, what doesn't, what's missing, and what's too restrictive, amongst other things. As the maturity of both the development practices and the Java technologies increases, the application architecture will need to be updated as well.

2 Overview of the Enterprise Architecture

SFA has a significant, ongoing investment in its current enterprise systems, and it realizes that it can't "untangle the hairball" overnight. As such, SFA is building a new enterprise architecture that facilitates three things:

1. Salvaging the investment SFA has already made in its legacy systems by building a layer that presents a unified view of the systems and their data
2. Implementing new systems that provide analytical tools that allow entirely new ways of analyzing enterprise data
3. Implementing a standardized application environment that will provide the application infrastructure for all new software development from this point forward

These items correspond to the Enterprise Application Integration (EAI) layer, the Data Warehouse layer, and the Internet Architecture layer, respectively. (See Figure 1.)

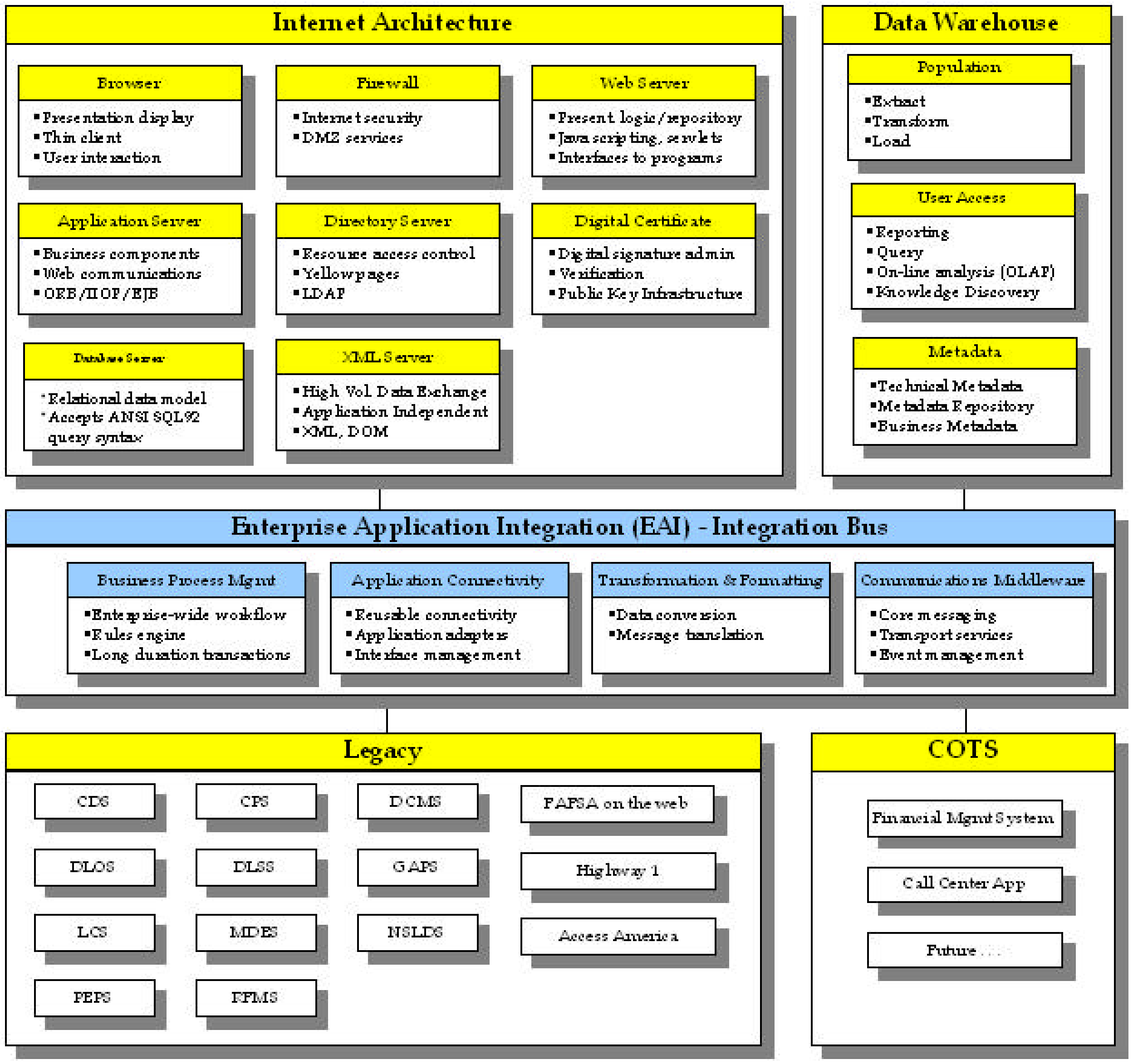

Figure 1: Enterprise Architecture

All communications between layers is mediated by the EAI layer. This allows for a unified view of the enterprise's data and system service resources, even as legacy systems are decommissioned and new systems are brought in to take their place.

3 Overview of the Application Architecture

SFA has decided to implement a standardized software development and deployment architecture for all new software development going forward. This is in contrast to the past practice of "anything goes," where software developers were allowed to use nearly any combination of hardware, operating system, communications protocols, and programming language they wished. While convenient for the software developers, this practice has saddled SFA with a complex, heterogeneous environment that is expensive to maintain and difficult to modify as business needs evolve. To break from this practice SFA has decided to adopt an *Internet-centric, multi-tiered, component-based* application environment.

Internet-centric:

All new custom-developed software -- and to the degree possible, all commercial, off-the-shelf software -- will utilize common Internet standards like TCP/IP, HTTP, SSL, HTML, and ECMAscript for communications and presentation. For applications that do not require a user interface, like business-to-business or e-commerce data exchanges, XML will be used. Other common standards like LDAP, X.509, and ANSI SQL '92 will also be used where applicable to provide services throughout the enterprise. (See the *SFA Common Operating Environment* document for the complete list of standards.)

Multi-tiered:

Application functionality will be distributed across several tiers. User interface presentation will be handled by thin clients like Web browsers or lightweight client applications. Static information resources are hosted on a Web server. Presentation logic, application-specific workflow, and reusable business logic will be encapsulated into components that are managed by an application server. Additional, special-purpose servers can be relied upon to provide enterprise-wide services for security, authentication, personalization, and business-to-business data exchange. Existing enterprise data resources will continue to be housed in a separate legacy layer that is accessed via the EAI layer. Storage for new data will reside in the Internet layer.

Component-based:

As much reusable business logic as possible will be encapsulated into components, which are independent objects that are dynamically and transparently executable over a network by multiple different applications simultaneously. These components reside on a dedicated server called an *application server*. The repository of components on the application server will allow new applications to rapidly and efficiently leverage earlier development efforts, facilitating a "plug and play" approach to building applications.

While there are several component standards to choose from, SFA has decided to adopt the Enterprise JavaBeans (EJB) component standard from Sun Microsystems. EJBs are part of a larger standard, known as the Java 2 Enterprise Edition (J2EE), which is based on the Java 2 Standard Edition (J2SE) language and API platform. J2SE specifies the Java language, a core set of APIs (including user interface, networking, database access, and security), and the Java Virtual Machine. The J2EE standard extends the J2SE with a set of APIs that provide uniform access to traditional middleware and infrastructure services, regardless of their particular implementation. This helps developers focus on developing the business logic of the application instead of having to also build common enterprise services into their software.

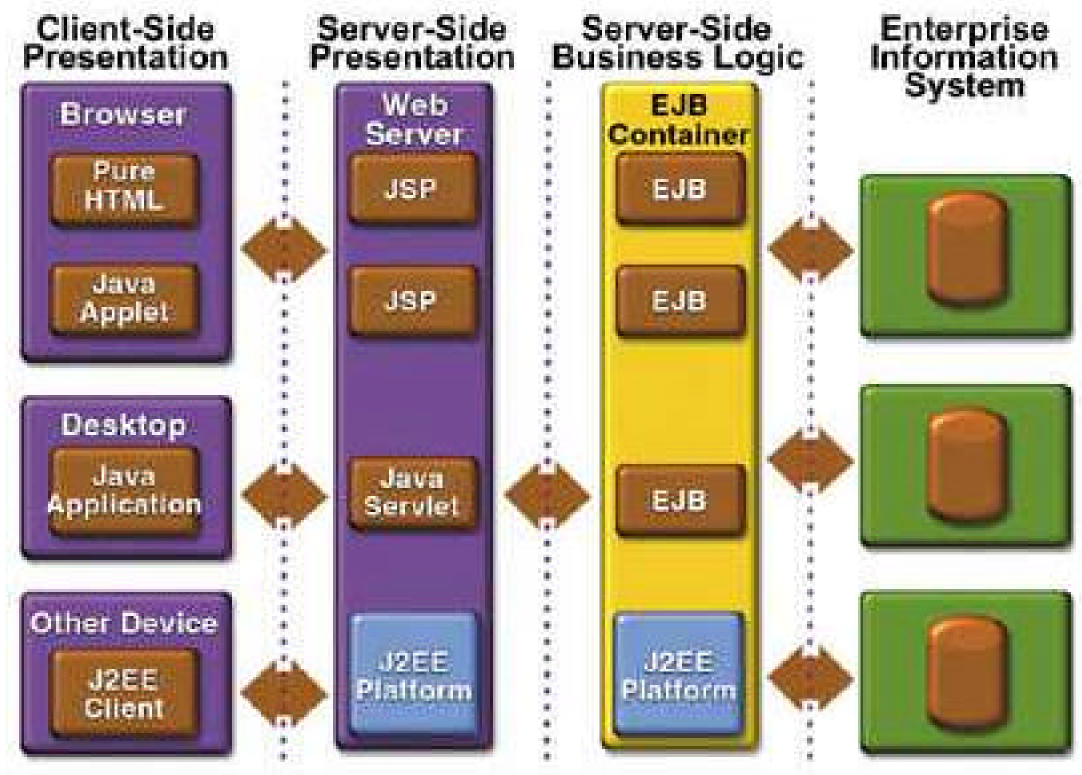

Figure 2: Generic J2EE Application Model

Figure 2, "J2EE Application Model," illustrates the typical layers and technologies that a J2EE application is distributed across. (Also possible, but not explicitly represented in this figure, is business-to-business data exchange using XML. An XML-enabled client would exchange information with a Java Servlet in the Server-Side Presentation layer over HTTP or HTTPS.)

4 Standards Covering All Aspects of Development

4.1 Java Coding Conventions

All developers will follow the naming and commenting conventions outlined in the Java Code Conventions documentation maintained by Sun Microsystems.

All custom-developed packages will utilize `gov.sfa` as their naming base. ***Note:*** *SFA does not, at this time, have the associated domain name (`sfa.gov`) registered with the GSA via <http://www.nic.gov/>. We recommend that SFA do so.*

4.2 Security

The SFA's standard method for securing TCP/IP-based transactions is called Secure Socket Layer (SSL). SSL establishes a framework in which a cryptographically-secure network connection between the browser and the web server can be made. This allows the client and server to exchange data without allowing others on the network to see the cleartext of the data transmitted, and it also ensures the reliability of the data -- what is sent is what is received. SSL can also be used to authenticate the identity of the client when the client is supplied with a digital certificate. SSL is not tied to any one particular form of data encryption; which encryption algorithm and bit-strength used to secure the data is negotiated between the server and the browser when the SSL session is established.

SFA must also utilize products that are compliant with the FIPS-140-1 standard for cryptographic security. (The legal authority for issuance of FIPS PUBS derives from Section 111(d) of the Federal Property and Administrative Services Act of 1949 as amended by the Computer Security Act of 1987, Public Law 100-235.) Every reasonable effort has been made to select vendors whose products have successfully completed a FIPS-140-1 audit, thus allowing application developers to rely on the audited mechanisms provided by the J2EE application development environment.

If the data being transferred between the application and the user is sensitive in nature, e.g., financial data, the data transfer will be cryptographically secured with SSL v3. The bit strength of the encryption to use (40-bit vs. 128-bit, for example) is left up to the judgement of the application developer, but a single application should use the same bit strength throughout all interactions that need to be secured. The preference, however, is to utilize 128-bit (or stronger) encryption where not a prohibitive burden upon the target user population. The recent relaxation of the export controls on products implementing strong encryption should facilitate the broader use of it in the future.
For data connections between components of an application, or between an application and enterprise-wide services like an LDAP server, SSL encryption will be used if the data is considered sensitive or if the transaction can potentially provide access to data considered sensitive.

4.3 Common Personnel Roles

The J2EE application programming methodology specifies several formal roles that are unique to component development and the EJB component methodology in particular. Each role may be performed by a different party, but a single party may perform several EJB architecture roles. For example, a single programmer may perform the two EJB architecture Roles of the Enterprise Bean Developer and the Application Assembler.

Application Component Providers produce the building blocks of a J2EE application. This role is further decomposed into the following skill areas:

HTML Specialists, Graphic Designers, Multimedia Authors

These are the individuals who are responsible for the implementation of the Web client user interface. They are skilled in HTML markup, graphics design for web sites, and Web-based multimedia technologies like Flash animation and streaming audio/video. They are not required to know any Java development beyond how to use any JSP custom tags created by the application developers. Their primary focus is on development of the Views in the MVC design. Their output is the set of HTML and JSP pages that make up the application's user interface, as well as the associated graphics and multimedia files.

Application Developers

Application developers produce the servlets and supporting libraries that implement the Controller components of the MVC design. As such, they integrate the Views with the Models and provide the associated workflow, error handling, etc. Application Developers are also responsible for the development of the JSP custom tag libraries that are used by the HTML specialists. The Application Developer is typically an application domain expert. The developer is required to be aware of the servlet environment and its consequences when programming, including concurrency considerations, and create the web application accordingly.

Enterprise Bean Developer

The Enterprise Bean Developer (EJB Developer for short) is the producer of enterprise beans. His or her output is an ejb-jar file that contains one or more enterprise bean(s). The EJB Developer is responsible for the Java classes that implement the enterprise bean's business methods; the definition of the bean's remote and home interfaces; and the bean's deployment descriptor. The deployment descriptor includes the structural information (e.g. the name of the enterprise bean class) of the enterprise bean and declares all the enterprise bean's external dependencies (e.g. the names and types of the resource managers that the enterprise bean uses). The EJB Developer is typically an application domain expert. The EJB Developer develops reusable enterprise beans that typically implement business tasks or business entities. The EJB Developer is not required to be an expert at system-level programming. Therefore, the EJB Developer usually does not program transactions, concurrency, security, distribution, or other services into the enterprise Beans. The EJB

Developer relies on the EJB Container for these services. An EJB Developer of multiple enterprise beans often performs the EJB architecture Role of the Application Assembler.

Application Assembler

The Application Assembler combines enterprise beans into larger deployable application units. The input to the Application Assembler is one or more ejb-jar files produced by the EJB Developer(s). The Application Assembler outputs one or more ejb-jar files that contain the enterprise beans along with their application assembly instructions. The Application Assembler has inserted the application assembly instructions into the deployment descriptors. The Application Assembler can also combine enterprise beans with other types of application components (e.g. JSPs, servlets) when composing an application. The Application Assembler is a domain expert who composes applications that use enterprise Beans. The Application Assembler works with the enterprise Bean's deployment descriptor and the enterprise Bean's client-view contract. Although the Assembler must be familiar with the functionality provided by the enterprise Beans' remote and home interfaces, he or she does not need to have any knowledge of the enterprise Beans' implementation.

Deployer

The Deployer takes one or more ejb-jar files produced by a EJB Developer or Application Assembler and deploys the enterprise beans contained in the ejb-jar files to the application server. The Deployer must resolve all the external dependencies declared by the EJB Developer (e.g. the Deployer must ensure that all resource manager connection factories used by the enterprise beans are present, and he or she must bind them to the resource manager connection factory references declared in the deployment descriptor), and must follow the application assembly instructions defined by the Application Assembler. To perform his role, the Deployer uses tools provided by the application server vendor. The Deployer's output are enterprise beans (or an assembled application that includes enterprise beans) that have been customized for the target operational environment, and that are deployed in a specific EJB Container. The Deployer is an expert at a specific operational environment and is responsible for the deployment of enterprise Beans. For example, the Deployer is responsible for mapping the security roles defined by the Application Assembler to the user groups and accounts that exist in the operational environment in which the enterprise beans are deployed.

The deployment process is typically two-stage:

- The Deployer first generates the additional classes and interfaces that enable the container to manage the enterprise beans at runtime. These classes are container-specific.
- The Deployer performs the actual installation of the enterprise beans and the additional classes and interfaces into the application server.

In some cases, a qualified Deployer may customize the business logic of the enterprise Beans at their deployment. Such a Deployer would typically use the container tools to write relatively simple application code that wraps the enterprise Bean's business methods.

In addition to the above roles there are several roles that are associated with any software development effort, but which are beyond the scope of this document. They include systems administrators, who are responsible for the operation and maintenance of the hardware and operating software that supports the application. Another common role is that of a source code manager who ensures the reliability and operation of the source code repository system.

4.4 Component Library and Reuse

Note: *The component library does not yet exist within the SFA. We recommend that SFA implement the library's basic architecture and functionality prior to the first use of the Internet development environment.*

The component library is a central Web site that catalogs all the major components that have already been developed and that are available for reuse in other projects. It also lists components that have been identified for inclusion in the library, but which are currently under development or which are scheduled and funded for development. The documentation is in standard JavaDoc format, providing the complete object signature as well as all public and remote interfaces. The validated range of inputs and outputs (behaviors) are also included, if appropriate, as well as a general history of the component (project that originally built the component, projects currently or forecasted to make use of the component, and the component's owner and maintenance contact person(s)).

The library facilitates component discovery through both browsing and searching. Components are categorized according to data and business logic domains as well as package organization, and can be browsed according to any one of these categorizations. A keyword search facility is also included.

The component library is a dynamic resource that is expected to undergo continual change as new applications are developed, business or legislative drivers change, etc. Part of the application development process is identifying opportunities to create reusable components and then making them available for reuse via the component library. General information about these components, once identified, should be provided to the component librarian via an on-line form. The librarian will then add this information to the section dedicated to components currently under development.

Once the development of a component is complete and it has been successfully deployed in a production environment the development team will provide the source code for the component so that it may be added to the library. If the component needs to be updated or removed from the library, the component's owner should contact the librarian.

5 Browser-Based Application Construction Standards

5.1 Model, View, Controller (MVC) Design Pattern

The standard architecture for the development of all applications is based on the Model, View, Controller (MVC) design pattern. *Models* are the enterprise data and business logic resources, and are typically implemented as EJBs. *Views* are static HTML templates that include JSP code to provide users with a dynamic view of the data and business logic contained in the EJB models. *Controllers* are servlets that mediate between the models and the views, providing request handling and workflow management within the application.

In order to aid the separation of the Views from the Controllers and Models, all system inputs are translated into *Business Events*. A Business Event is an application-centric, view-independent representation of a user request. The user request, in turn, is a representation of a business function. The catalog of Business Events that an application is able to process becomes the functional capability of the application itself.

MVC provides a useful separation of application responsibility and development team duties, allowing each member of the application team to focus on their particular skillset. For example, Web designers can focus on developing Views (HTML pages) without having to know about Java software development; they only need to learn a few new HTML/XML tags to be able to display the properties of the EJB Models. Meanwhile, junior developers can focus on implementing Controllers in pure Java without having to also develop the HTML look-and-feel of the application. Lastly, senior developers can focus on creating the Models of the enterprise's data architecture and business processes in a reusable and scalable fashion.

5.2 Event Flow

Figure Three shows the event flow for a typical MVC-based application. All requests from the client go to a servlet known as a *Front Component*, which provides a single point of entry for all application interaction. The requests are then sent to another servlet, known as the *Request Processor*, which converts the client request (in this case, an HTTP POST or GET) to a Business Event. The Business Event is then sent to the *Web Controller* servlet. The Web Controller sends the Event to the *EJB Controller* servlet, which processes the Event. Depending on the Event, the EJB Controller will coordinate the execution of business logic and data manipulation by one or more EJBs. Once the Event has finished processing, the EJB Controller notifies the Web Controller if the Model data has changed, necessitating an update of the Views' data JavaBeans. If it has, the Web Controller passes the update event to the data JavaBeans (1), which then request updated data from the EJBs. The Web Controller then selects the appropriate JSP response template (View) to return to the client (2). The JSP template collects whatever properties it needs from the data JavaBeans when it is executed by the JSP Engine, and the resulting HTML is passed back to the client.

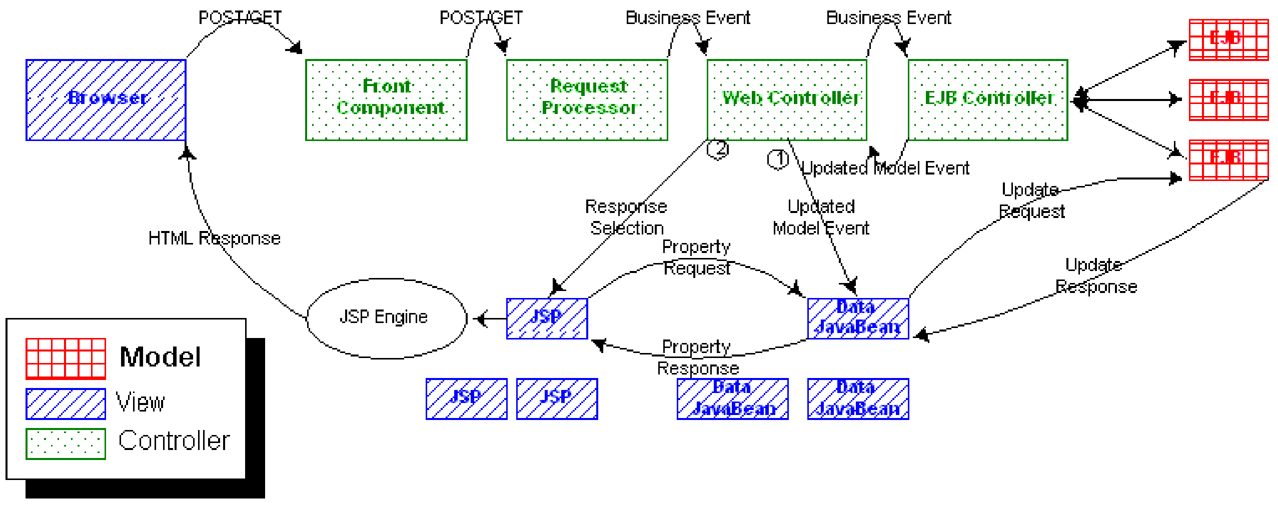

Figure 3: Detailed Application Model

In Figure Three, the Model components are in red, the View components are in blue, and the Controller components are in green.

Let's imagine that a student wants to make a payment on a student loan, and that he's decided to use one of the new applications the SFA has developed with this architecture to make this possible over the Web. Let's also assume that the student has already authenticated himself, that his browser has established a cryptographically-secure network connection with the SFA's Web server, and that he's looking at a page that is requesting payment information. After entering his payment information he clicks a button labeled "Send Payment."

The "Send Payment" button causes the browser to initiate an HTTP POST connection with the Web server, transferring the student's payment information to the Front Component servlet. The Front Component does some simple security and request validation before passing the POST data to the Request Processor servlet. The Request Processor evaluates the POST data, determines that a client (the student) is initiating a `makePayment` Business Event, and passes the `makePayment` event to the Web Controller servlet.

The Web Controller performs several actions after receiving the `makePayment` event. First, it passes the `makePayment` event to the EJB Controller, which is responsible for managing the EJBs that actually execute the business logic of the Business Event. Thus the EJB Controller would know that a `makePayment` event needs to:

1. Get the funds from the student's fund source (credit card or checking account)
2. Deposit the funds into the account of the loan holder
3. Update the account record with the transaction details

Each one of these steps will likely be handled by one (or more) EJBs. If there are any errors along the way the EJB Controller will usually pass the exception back to the Web Controller so that the Web Controller can send an error message to the client. Otherwise, the EJB Controller will complete successfully and, if there were changes to the Model data, pass an Updated Model Event back to the Web Controller.

The Web Controller, upon the EJB Controller's completed processing of the Business Event, first propagates the Updated Model Event (if it occurs) to any existing data JavaBeans in the View. These data Beans mirror the data contained in the entity EJBs present in the Model, but in a format that is consistent with and meaningful to the application's problem domain instead of the enterprise's data architecture. (The two data models should be very similar, but an application frequently needs to only manipulate a small subset of the entire data model; thus the local mirror is typically a simplified version that's tailored to the application's problem domain.) The data Beans know how to interrogate the entity EJBs directly so that they can update themselves when they receive the Updated Model Event notification.

The second task of the Web Controller is to select the appropriate display template, based on the result of the EJB Controller's processing of the Business Event. Assuming the student's payment transaction completed successfully, the Web Controller selects a JSP template that indicates such, along with a listing of the account transactions that have occurred over the past month -- including this latest payment transaction. The Web Controller reads the JSP template from disk and sends it out to the client via the JSP Engine as the application's response to the client's original POST request. The JSP Engine interprets the JSP commands that are embedded in the JSP template. In this case, the embedded JSP happens to be a custom JSP tag that gets the list of the past month's account transactions from a data Bean. The custom tag evaluates to a nicely formatted HTML table of the account transactions, which gets set to the Web client along with the static HTML in the JSP template.

5.3 Implementing Models with EJBs

The EJBs that make up the application's Model can be implemented as either Session Beans or Entity Beans, as is appropriate for their function.

The Model is an abstraction of the data on which an application is based. Model data should only be updated when events that require data changes are passed to the EJB Controller, and the EJB Controller should be the only component to actually modify the Model's data. After each event is processed the EJB Controller is responsible for returning an Updated Model Event to the Web controller if an update has occurred. Model data should only be modified by the EJB Controller; it can be directly read by view components, but never written.

The EJBs implementing the Model will interact with the other layers (legacy, COTS, and data warehouse) via the Enterprise Application Integration Bus and the CORBA interfaces it provides. The CORBA interfaces, which have not been developed yet, will provide a unified view of enterprise data and business logic. Internet developers will use the JavaIDL tools to compile the IDL provided by the EAI Bus developers into the necessary Java stubs. At no time will any Model component make a direct call to a component or service in another layer.

If a new application needs persistent storage of data that does not already exist elsewhere in the enterprise, that data may be stored in the Oracle database that is a part of the Internet layer. All interactions with this data storage service will be done via JDBC 2.0 (the `java.sql` and `javax.sql` packages).

5.4 Implementing Views with JSPs

JSP-based Views are HTML pages that have the ability to expose dynamic enterprise data via JSP tags that get information directly from a JavaBean. These JSP tags are interpreted at response time by the JSP container, and the result is substituted in place of the JSP tag. Thus they are a specific representation of the model for a client that can understand HTML. The JSP tags, which are similar to HTML tags, are readily learned by HTML page designers. Given a copy of the data model, an HTML page designer can display any application data that is contained in a JavaBean.

For example, let's say that a designer wants to display the current year. To gain access to the standard JavaBean called `calendar` that represents the current date, the designer would include the following tag at the beginning of the HTML page to create an instance of the `calendar` class:

```
<jsp:useBean id="today" class="calendar.jspCalendar" />
```

Then, wherever the designer wants to put the date on the page, she'd use the following tag:

```
<%=today.getYear() %>
```

In the resulting HTML page that is sent to the requesting browser, the entire JSP tag is replaced with the value returned from the `getYear()` method call performed on the today `object`.

For more complicated dynamic information, application developers will need to develop a custom tag library that the HTML page designers can take advantage of. For example, a query to see all the payments made on a student loan in the current year can result in an arbitrary number of rows in the result set. The designer needs to be able to programmatically iterate over the result set, displaying each line in turn. Instead of forcing the HTML designer to know Java, or having a Java developer edit the HTML files, a Java developer can create a set of custom JSP tags that display a table containing the payment information. To use these tags the designer first declares that she is going to use a custom tag library:

```
<%@ taglib prefix="myCustomTagLibrary"
uri="/myApplication/myCustomTagLibrary.tld" />
```

Then she'll usually instantiate the objects in the library by calling a startup routine:

```
   <myCustomTagLibrary:defineObjects />
```

Finally, she can use the custom tags, like this:

```
   <myCustomTagLibrary:getLoanPayments studentID="123-45-6789" year="1999" />
```

In the resulting HTML page that is sent to the requesting browser, the entire custom tag is replaced with a table containing a list of all the loan payments in 1999 from student #123-45-6789.

5.5 Implementing Controllers with Servlets

All Controller components should be implemented as servlets.

The **Front Component** is a central servlet that receives all HTTP requests. It ensures that all necessary Web components needed by the application are initialized at the correct time and that all HTTP requests are sent to the Request Processor. Front Components are useful because they provide a single entry point to an application, thus making security, application state, and presentation uniform and easier to maintain.

A **Request Processor** is the link between the Web application and an HTTP-based client. The Request Processor is responsible for converting an HTTP request to Business Events that will be used throughout the application. This component allows the application developer to centralize all HTTP-specific processing in one location. This component also allows the EJB portion of the application to be independent of any single client type.

The **Web Controller** is responsible for forwarding the events generated by the Request Processor component to the EJB Controller. The Web Controller ensures that the resulting Updated Model Events returned from the EJB Controller are propagated to the appropriate data JavaBeans components.

An **EJB Controller** accepts events that are received from the Web Controller and makes the necessary calls on the enterprise beans that are affected by the event. The EJB Controller is also responsible for maintaining the state of the user session with the application. After each event is processed the EJB Controller is responsible for returning an Updated Model Event to the Web Controller.

5.6 Client-Side Presentation

5.6.1 Graphics Formats

All graphics will be in either GIF 87a format or JPEG format. The GIF format is preferred for the majority of graphics, and a graphic in the GIF format will always end it's filename with `.gif`. No preference is given between interlaced and non-interlaced GIF graphics.

If the graphic is photographic in nature, meaning that there are many subtle variations of a particular shade, the JPEG format is frequently a better choice. There are a variety of compression levels available when using the JPEG compression scheme. The particular level to use is left to the discretion of the developer; however, image quality should not be sacrificed for size.

If a graphic is referenced by an HTML `img` tag, the `height` and `width` attributes must be included in the `img` tag and they must match the actual dimensions of the image being referenced.

If the graphic is going to be used as part of an application that is to be used by the general public, only the "browser-safe" colors are allowed. This is a palette of 216 colors that every browser that displays on an 8-bit color display device can render without resorting to dithering. Most modern graphics programs include tools for restricting a graphic's color palette to these colors.

5.6.2 HTML, JavaScript Standards

In general, the philosophy that drives the HTML coding recommendations is one of "universal design." This means that pages will be designed to look good across all browsers and that browser-specific versions of pages will not be developed. The architect must determine which one of three possible audiences will be the primary users of the application: the general public, business partners, or SFA employees.

- For sites that are accessible to the **general public**, the highest level of markup allowed is compliant with HTML 3.2, plus frames. Browser-specific tags and/or attributes are expressly not allowed. JavaScript, if it must be used, should be used sparingly and should be compliant with v1.2 of the language. Architects can not rely on the proper operation of the JavaScript, however. All data received from the client must be validated on the server side in addition to any JavaScript validation done on the client side. The pages that utilize JavaScript must be usable even if the JavaScript does not function correctly.

- For sites that are restricted to **business partners**, the highest level of markup allowed is compliant with HTML 4.0, but cascading style sheets and layers are not allowed because of their generally poor implementation in current browsers. JavaScript that is compliant with v1.4 of the language is allowed. All data received from the client must be validated on the server side in addition to any JavaScript validation done on the client side.

- For sites that are restricted to **SFA employees**, the highest level of markup and JavaScript allowed is compliant with the highest level of HTML that the SFA's standard browser supports. Data validation must still be performed on the server side in addition to any JavaScript validation done on the client side.

Architects should note that there may be other standards, particularly concerning Web site graphic design, to which they may also be subject to.

Regardless of the application's audience, the following standards also apply. Filenames of all HTML documents will end in `.html`. Spaces or other characters that must be URL escaped are not allowed in the filenames. Every HTML file will include the appropriate DTD declaration as the first line of the file, and the file will be validated with a true SGML validation program. Currently, the Microsoft Windows platform has only one true validation program, called "A Real Validator." It is available from <http://www.arealvalidator.com/>. The W3C offers a free, on-line validator at <http://validator.w3.org/>. Other validators must be approved prior to their use. All HTML documents that contain server-parsed elements (like JSP) should be validated after they are served from the Web server, thus checking the document in the final form that users will see.

5.6.3 JSP Standards

Since the design focus for the SFA's use of JSP technology is to provide a frame in which to display enterprise data, the use of complex JSP scripting in an HTML page should be minimal to non-existent. For the vast majority of applications, access to Bean properties and some custom tags that implement simple looping constructs should be sufficient. In particular, the JSP templates should not need any embedded Java.

6 Java Applet Construction Standards

Much of the philosophy behind the preceding standards discussion regarding client-side presentation is applicable to the inclusion of Java applets in Web pages. In particular, it is necessary to segment the potential user population based on their browser's capabilities and to then target the applet's development to those technology constraints.

- For sites that are accessible to the **general public**, applets are forbidden. The delays experienced when downloading an applet and initializing the browser's Java Virtual Machine (JVM) are frustrating to the user. In addition, because of the generally low quality of the JVMs embedded in the browsers most commonly in use across the Internet, extensive testing (greater than a dozen possible combinations of operating systems, browsers, and JVM versions) and coding around implementation quirks are cost prohibitive. Finally, applets are generally incapable of providing alternative forms that are able to provide access to disabled users.

- For sites that are restricted to **business partners**, applets may be used if necessary, but they aren't encouraged. It is strongly suggested that developers use the downloadable Java Plug-In product with the Java Runtime Environment v1.2 for execution consistency and guaranteed access to Java 2 functionality. Architects should carefully consider this barrier to entry for business partners who wish to use the software, however.

- For sites that are restricted to **SFA employees**, applets may be used if necessary. It is strongly suggested that developers use the downloadable Java Plug-In product with the Java Runtime Environment v1.2 for execution consistency and guaranteed access to Java 2 functionality.

7 Java Application Construction Standards

Standalone Java applications, if developed, will be based on the Java 2 Standard Edition APIs. Java applications will not take advantage of other APIs unless they are 100% Java, and even in that case those APIs should not be used if the core or extended APIs are suitable for the task.

As with the Web-based applications, Java applications will utilize the MVC design pattern architecture.

They will utilize the Java 2 Swing UI components with the Pluggable Look and Feel set to the Java Look & Feel. Accessibility features offered by the Accessibility API must be utilized to provide a complete user experience for those who need to utilize adaptive technologies.

8 Faceless Application Construction Standards

"Faceless" applications are those applications that are meant to facilitate automated business functionality without any direct user involvement. An example of this kind of application is a servlet that receives an XML document, parses it, updates a database with data from the XML document, and returns a confirmation message to the original sender. Thus they can be considered to implement (and be subject to the architecture standards relevant to) the Model and Controller aspects of the MVC design pattern.